A comparison of two SPLE tools: Pure::Variants and Clafer tools

Miika Oksanen

Helsinki May 21, 2018

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Miika Oksanen

Työn nimi — Arbetets titel — Title
A comparison of two SPLE tools: Pure::Variants and Clafer tools

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	May 21, 2018	58 pages

Tiivistelmä — Referat — Abstract

In software product line engineering (SPLE), parts of developed software is made variable in order to be able to build a whole range of software products at the same time. This is widely known to have a number of potential benefits such as saving costs when the product line is large enough. However, managing variability in software introduces challenges that are not well addressed by tools used in conventional software engineering, and specialized tools are needed.

Research questions: 1) What are the most important requirements for SPLE tools for a small-to-medium sized organisation aiming to experiment with SPLE? 2) How well those requirements are met in two specific SPLE tools, Pure::Variants and Clafer tools? 3) How do the studied tools compare against each other when it comes to their suitability for the chosen context (a digital board game platform)? 4) How common requirements for SPL tools can be generalized to be applicable for both graphical and text-based tools?

A list of requirements is first obtained from literature and then used as a basis for an experiment where support for each requirement is tried out with both tools. Then a part of an example product line is developed with both tools and the experiences reported on. Both tools were found to support the list of requirements quite well, although there were some usability problems and not everything could be tested due to technical issues. Based on developing the example, both tools were found to have their own strengths and weaknesses probably partly resulting from one being GUI-based and one textual.

ACM Computing Classification System (CCS):
(1) CCS → Software and its engineering → Software creation and management → Software development techniques → Reusability → Software product lines
(2) CCS → Software and its engineering → Software notations and tools → Software configuration management and version control systems

Avainsanat — Nyckelord — Keywords
Software product line, SPLE, tools

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Software product line engineering (SPLE) is a paradigm of software development that emphasizes systematic, large-scale reuse over a range of software products with similarities. Instead of focusing on individual products and reusing their development artefacts in other products when possible, in SPLE the focus is on a range of software products, i.e. the *software product line*. SPLE has been successfully applied in many organizations for decreasing costs, improving software product quality, and reducing the time to market [VdLSR07]. Other potential benefits of adopting SPLE include improving cost estimation for software products and uniformity of the user interfaces of the products [PBvDL05].

To maximize reuse, products are planned and designed to include common code whenever and as much as is feasible. Typically this is done by creating a common platform that forms the base architecture for all the products in the PL, and attaching variable parts to it depending on the specific product being built [PBvDL05]. The variable parts may be common to multiple products or they may be product-specific when that makes the most sense [VdLSR07]. The potential benefits of SPLE are highly dependent on successfully designing and making use of this kind of extensible or variable architecture.

Designing and managing the product line requires some specialized modeling techniques in addition to those needed in software engineering in general [PBvDL05]. These are referred to as *variability modeling* techniques [SD07]. They are typically not needed in single system engineering. The model of the variability, i.e. the *variability model*, has to be used for building both the software architecture for the PL and the individual products. Furthermore, both the variability model and the software architecture based on it need to be maintained, especially since software product lines tend to be long-term commitments. It has been estimated that the additional overhead, effort and complexity that come with shifting to the SPLE approach typically start to pay off after from three to ten products [VdLSR07], when compared to developing single products at a time.

Building and managing software development artefacts that include variability information can be complex and would be very hard and time-consuming to do without automation provided by software tools. Existing tools that are not specifically made to provide support for SPLE are generally not sufficient for all its needs. Therefore, specialized SPLE tools are needed. However, at least until recently, it has been fairly

common among industry practicioners to make their own tools or to use existing, domain-specific or even application-specific tools [BRN+13]. Creating one's own SPLE tool is a complex, time-consuming and costly procedure, while there might be no domain-specific tool that is suitable for a given product line. Thus, there is demand for more generic SPLE tools as well.

Since SPLE is concerned with the whole development process of a family of software products, the list of potential requirements for a SPLE tool is quite long. The matters are further complicated by there being a large number of both different approaches to SPLE and different variability modeling methods. Although there has been some work on SPL standards [CL16], the range of tools and related methods remains highly fragmented.

The research questions of this master's thesis are the following:

- **RQ1**. What are the most important requirements for SPLE tools for a small-to-medium sized organisation aiming to experiment with SPLE?
- **RQ2**. How well those requirements are met in two specific SPLE tools, Pure::Variants and Clafer tools?
- **RQ3**. How do the studied tools compare against each other when it comes to their suitability for the chosen context (a digital board game platform)?
- **RQ4**. How common requirements for SPL tools can be generalized to be applicable for both graphical and text-based tools?

Pure::Variants and Clafer tools are reasonably scalable and rich in functionality, and thus should have many prerequisites for being suitable for professional SPLE. Additionally, both tools are available for evaluation for free, have documentation available online, and are well researched and continously developed. Although there have been many evaluations and comparisons of SPLE tools [PCF15][EDDT11] [CPP+16][PSF+13][DSF07], few if any of them compare Pure::Variants and Clafer tools. The aim is to describe support for various use cases in more depth than in a binary fashion as in some other work, and to include a more complete set of requirements for this type of situation than in other work. Additionally this thesis aims to contribute to tool research and development by pointing out things that could be improved in both studied tools. Furthermore, there have been few if any comparisons of text-based tools such as Clafer tools against mostly graphical tools such as Pure::Variants.

The rest of the thesis is organized into five parts. The second one, section 2 is a literature review, providing additional background information and identifying things to look for in the tools and showing how the tools can be compared. The third, section 3 describes the methods, context and limitations of both of the experiments that were done for this thesis. The fourth section, section 4 describes conducting the experiments themselves and reports any relevant findings for both. In subsection 4.1, a generic evaluation and comparison of the tools is performed based on requirements found in literature, as identified in section 2. The different parts of section 4.1 deal with categories of specific tool functionality or other requirements. The categories are feature addition, constraints, attributes, model validation and error checking, inheritance and modularization techniques, and creating and editing configurations. Here the aim is to be as objective as possible and simply list what functionality is supported in each tool and how. In subsection 4.2, a small part of an imaginary digital board game platform SPL is made with both tools, and a small-to-medium sized organization that is trying out SPLE is assumed as a context. Subsection 4.2 is divided into parts that deal with specific work or tasks; installation of the tools, building the model, and product configuration. Here the findings will be more subjective than in the previous section, but possibly more relevant to the chosen specific context. As a final subsection of section 4, 4.3 summarizes findings of both experiments in the form of two tables. The two final parts are "Discussion" (5) and "Conclusions" (6).

2 Literature review

The first of the two parts of this literature review (2.1) aims to provide an overview on literature about product line engineering in general and independently from the SPLE tools. The second one (2.2) focuses on the tools.

2.1 Software product line engineering

Section 2.1.1 provides an overview of different variability modeling techniques. Additionally it contains background knowledge found from literature about feature modeling, which is the technique used in this thesis. Section 2.1.2 describes the SPLE process as it is defined in literature, but in broader terms. Section 2.1.3 goes into how SPLE is used in industry based on research reports and other literature. 2.1.4 presents some challenges related to applying SPLE.

2.1.1 Variability modeling

Software product line engineering (SPLE) introduces various activities and techniques that are not present in conventional or traditional software engineering [VdLSR07]. Since the focus on variability of the software products in the product line is one of the defining features of SPLE, modeling and managing the variability is central to applying SPLE [PBvDL05]. *Variability modeling* is a collection of activities and techniques related to coming up with a representation of commonalities and differences between the individual products, and making use of that representation [CGR+12]. When two or more products share functionality or non-functional properties, they can generally be partly implemented using shared design and implementation artefacts and tested with the same tests. The benefits of the SPLE approach are a direct result of this kind of maximized, planned reuse [VdLSR07].

There are numerous different variability modeling techniques and methods [SD07]. Some of the most popular ones include feature modeling (FM), orthogonal variability modeling (OVM), UML extensions, decision modeling, and domain specific languages [CB11][EKS13][BRN+13]. For most methods there are many different variants, and approaches that combine or integrate two or more methods exist as well [RFBCRC09]. This Master's Thesis focuses on FM and feature-oriented methods in general. This is partly because FM is the most widely researched coherent collection of variability management methods for SPL [CB11]. Moreover, feature

models are the most popularly used models of variability both in industry and SPLE research [BRN+13][CB11].

A *feature model* (FM) is a model of the whole SPL that focuses on the features of the software products. In this context, a feature can mean a piece of functionality, a non-functional property or any other characteristic measurable or observable from the final products [EBB05]. The FM is organized as a tree of features, where the root consists of the core platform architecture that is common for all the products in the product line [PB12] . The variable features are the other nodes of the tree [SBSQ11]. Feature modeling as a variability modeling method has its roots in *feature oriented domain analysis* (FODA), and all feature modelling techniques are basically extensions of FODA [CHE04]. In FODA, a feature can be *mandatory*, *optional* or *alternative*. A mandatory feature is included whenever its parent in the tree is included, while an optional feature can be included when its parent is included, depending on the product, and exactly one from a set of alternative features is included when its parent is included. In case where inclusion or exclusion of one feature needs to depend on other features' inclusion or exclusion in a way which cannot be inferred from the tree structure, cross-tree constraints can be defined.

There are many feature modelling approaches that extend FODA. Cardinality-based feature modeling (CBFM) is one widely researched FODA extension [SBSQ11][SD07] [EKS13]. It introduces the possibility to define a minimum and maximum number of features that must be selected from a feature group [CHE04]. Secondly, it enables including many copies of a single feature in a product. Thirdly, CBFM has support for features with *attributes* as well. Attributes enable a feature to be associated with a variable type such as integer or text, and then later including the feature with a chosen value for the individual product being assembled. Lastly, CBFM includes support for *feature diagram references*. With them it is possible to include a reference to a separate feature diagram (FD) or to a sub-diagram of the same FD in place of a feature. References to sub-trees of the same FD enable reusing parts of the model in other parts, while still keeping the tree-structure of the diagram. References to another FD enables modularization of the FM, which can help with scalability of the model [CHE04].

In order to be able to model concepts and properties that would be hard or complicated to model through basic feature modeling, *meta-models* can be used [DSF07]. In SPLE, meta-models are models that complement standard variability models and use semantics that are generally specific to the product line or the ecosystem

of product lines [RGD10]. A software product line meta-model can be thought of as specialization of FODA (or another variability modeling technique) that is custom tailored for dealing with issues specific to a given SPL, instead of going with more generic model semantics [BDA+14]. Meta-models can deal with low-level or specific details of the product line such as ways to connect pieces making up the SPL architecture.

2.1.2 The SPLE process

Pohl et al and Linden et al define SPLE as consisting of two sub-processes, *domain engineering* and *application engineering* [PBvDL05][VdLSR07]. Domain engineering is mainly responsible for planning, defining and designing the whole product line. Application engineering on the other hand deals with all the specific products. In both sub-processes, requirements are defined, architecture modelled and managed, software components implemented, and software tested. The difference is that domain engineering focuses on parts of the software that are designed to be common for many products, whereas in application engineering it is on concrete, whole products. The sub-processes are generally thought to be separate, often with different personnel carrying out their activities. Typically they happen to some extent simultaneously, and application engineering uses and provides feedback for the design and software artefacts created in domain engineering.

After at least a partial plan for the range of products in the SPL has emerged, the domain engineers create requirements for the SPL [VdLSR07]. The requirements are used for coming up with a model of its variability using some variability modelling method such as ones discussed in the previous section. The *SPL architecture* consisting of the common and variable parts for all the products in the SPL is designed based on the requirements, with keeping the goal of maximising reuse in mind [PBvDL05]. The parts of the software that are known to not be common to the whole SPL, but that are still known or likely to be common to several products are included in the SPL architecture. If FM is used, the variable parts of the SPL architecture can be modelled with non-mandatory features (or their sub-features) [VdLSR07]. Constraints can be used to eliminate products that do not make sense or are unnecessary. If a component or a part of software architecture is considered to be needed only for a single product, it can be left as the concern of application engineers. Then the variability model for the SPL architecture is validated, and the architecture itself is implemented. Depending on the used testing strategy, tests can

be developed as a part of domain engineering to be reused in application engineering, or they can be done separately in both or even only in application engineering [NdCMM+11].

Application engineering is responsible for configuring and building the actual products. The product-specific requirements are filled by taking the SPL architecture provided by domain engineering, including some specific configuration of the core assets and then including product-specific assets when necessary [VdLSR07]. This process is guided by the variability model provided by domain engineers. All the development artefacts that are specific to some product can either be built and maintained by application engineering or a decision can be made to include them as parts of the SPL architecture. Product configurations need to be validated in terms of what is allowed by the variability model designed by domain engineers [VdLSR07]. All the dependencies and parent-child relations between features have to be met. In addition, any constraints have to hold as well. Testing for the product-specific components and the complete products is performed in application engineering as well [NdCMM+11].

The point in time at which a specific feature or variant is actually realised and after which it is no longer variable can differ, and is referred to as *binding time*. The binding time can be during design, in which case the selection of a component providing the chosen feature is finalized during design [CSD07]. Other options include compilation time, dynamic configuration during run-time, and having the software read a configuration file when starting up.

Typically all the activities in both sub-processes either overlap to some extent or are performed in a cyclic fashion, a part being done in each cycle [VdLSR07]. This is because product lines tend to be rather long-term commitments, and during their lifetime requirements are likely to change or evolve. Furthermore, new products using old product-specific components can be planned, and in those cases it might make sense to make those components a part of the SPL architecture. Communication between the sub-processes and feedback from application engineering to domain engineering is important.

2.1.3 Industry practices

In practice, SPL development takes either a *proactive*, *reactive*, or *extractive* approach [ANAV10]. In the proactive approach, the emphasis is on planning the

whole range of products first, and then proceeding to modeling the variability of the whole PL. In a purely reactive approach, on the other hand, the PL is first designed and built with only one product. The PL can then later be extended by adding more products into it. An extractive approach refers to taking some existing software that was not built with SPLE methodology, and then *extracting* a product line from it with some techniques. All these approaches are commonly used in industry [BNR+14], with extractive being the most popular and proactive the least popular one [BRN+13], even though research in variability modeling techniques for SPLE has focused mostly on the proactive approach [ABMG12][ANAV10].

The number of features of the models used in industry can range from tens to thousands. There is often effort to keep feature models simple by keeping the tree depth small and not using many dependencies or constraints [BNR+14]. It often may not be worth the effort to include all available knowledge about the products directly in the models, i.e. making the model *complete* [SRM11]. Variability modeling is sometimes used only for managing and communicating product plans and requirements without automated product generation from the models. The models are often managed in a completely centralised fashion. The extent of which constraints are used may be dependent on the application domain [BNR+14].

2.1.4 Challenges

Including the right amount of detail in a feature model can be challenging. A part of a FM can have different degrees of granularity, meaning how much is included in single features. Making multiple feature models that have hierarchical relationship with each other or grouping features could help with this, but this can require significant domain knowledge [BNR+14]. Other factors that need to be considered when deciding the FM granularity include feature stability and team structures [DFE14]. Another challenge is managing evolution due to changed requirements and changing technology. It can be hard or at least costly, as each change may affect many parts of the models [SRM11].

Fitting requirements engineering techniques in SPLE, especially with feature-oriented methods can be problematic. Different feature modeling techniques found in SPLE research are compared in terms of their support for requirements engineering at both domain and product level in a study by Asadi et al [ABMG12]. Support for non-functional requirements and validation and verification of requirements was found lacking in the reviewed techniques. Additionally, there was poor or inexistent

support for including stakeholder preferences in the requirements models. Another, related issue is that even though feature models were originally meant to be used by customers themselves for picking features for products, the customers may not possess the knowledge and skill needed [BNR+14].

The canonical SPL process generally described in literature requires a lot of up-front planning and modeling effort. For some organizations - especially small and medium-sized ones - an approach more in line with agile methods might work better [BE13a][DSNO+14]. There are some approaches that aim to improve a specific part or aspect of SPLE through inclusion of agile methodology, like agile product derivation [OMTR12], while some approaches take an agile process and include SPL methodology into it, as in a study by Bagheri et al [BE13a]. The agile SPL approaches are highly varied and each of them highly specific, and are not well or at all supported by some general tools commonly used in industry such as Big Lever GEARS and Pure::Variants [Ber12]. For these reasons, the agile approaches are mostly left out of scope of this work.

2.2 SPLE tools

Section 2.2.1 lists some functionalities that are needed or useful for supporting SPLE projects, while 2.2.2 looks into other factors that might affect tool choice, such as non-functional properties and suitability for different kinds of contexts for applying SPLE. Sections 2.2.3 and 2.2.4 give a brief overview on Pure::Variants and Clafer Tools based on manuals, user guides and other literature. 2.2.5 describes some studies where different SPLE tools have been evaluated and compared.

2.2.1 Useful functionalities and other characteristics

As mentioned in previous sections, the focus in this work is on feature modeling, so support for a modeling method similar or based on FODA is expected from a tool. However, basic FODA cannot handle more complex relations of features and dependencies between them, so support for an extended version of it may be needed [CSD07]. Automatic analysis is useful for detecting mistakes and anomalies such as dead features, i.e. features that are unselectable [RGD10][KAT16]. Automatic generation of valid configurations is often useful as well, although in practice, models may not be complete, and thus incorrect or useless configurations will also be generated [SRM11]. Some efficent way to filter out the incorrect configurations might

be needed. Another useful, related functionality is automatical completion of a partially made configuration as in [ABM+13]. Traceability from higher level assets such as architectural model to lower-level assets such as code and traceability of the feature model itself is also usually considered something that a tool should support [DSF07][RGD10] . It is especially important for managing changes and evolution of the SPL and the single products [LVV04].

A study by Berger et al [BSL+13] points out three additional techniques related to feature modeling that are important for scalability. They are default features, visibility conditions and derived features. Default features means that features can be defined so that they are included or not by default, which can save time during product configuration. Derived features are features, for which inclusion is calculated by some formula based on the model. Visibility conditions determine the visibility for a given feature. When a feature is invisible, it cannot be seen or even changed. Default features and derived features can be combined to come up with derived defaults. For derived defaults, only whether the feature is included by default is derived in this way, but their inclusion can be changed. All of these are commonly used at least in the systems software domain, according to Berger et al [BSL+13]. In a study by Savolainen et al [SBKM09], default features are leveraged for help with managing PL evolution. In this method, features go through a lifecycle of states that affect their default inclusion or exclusion.

Support for detailed modeling related to architecture or non-functional properties can be useful. Meta-modeling (see 2.1.1) support by the SPL tool can provide a way for this [DSF07]. Different FODA extensions or variants or some combination of them can meet all these needs, but this can increase the complexity of the feature model and the modeling language itself [BDA+14]. It should be noted that including the architecture model into the feature model requires a higher degree of completeness from the FM [SBKM09]. Information added to a meta-model could be used by the tool for optimization of quality properties as in a study by Antkiewics et al [ABM+13] for example. Nevertheless, building the meta-model and maintaining it takes time too, so it may not always be preferrable over some extension of FODA. In addition to support for basic feature modeling, traceability and product derivation, a work by Steger et al [STB+04] mentions documentation of architecture, and documentation, checks and other management for feature interfaces as requirements for a tool at Bosch Gasoline Systems. At the time of the study, no tool seemed to support everything needed, and they had to build their own tool, which ended up having poor usability. This is fairly common in other industry case studies as

well [VdLSR07].

Flexibility is needed to support different types of activities and different stakeholders with varying skills. One aspect of this is visualization. Variability can be presented visually with different views, such as a tree, a graph or a matrix. Each have their drawbacks and strengths. Pleuss et al [PRB11] compared and listed a large number of different views meant to be used during product configuration. Support for different views and switching between them can be useful. The tree view of a feature model is a natural way to visualize it, but scalability quickly becomes a problem when the model grows. Categories or groups of features can help with scalability by enabling the user to select a category or group to focus on at a time. Freely selecting any branches of parts of the model is useful for this as well [CSD07]. Communication and collaboration between different types of stakeholders is important in SPLE, and different kinds of visualizations with varying levels of abstraction and different focus provided by the tools can help with this [LVV04][DSF07]. Different stakeholders can also have preferences for prioritization of non-functional properties, and they might want to define constraints on the usage of resources they are responsible for [ASG+14]. Accounting for different types of stakeholders is important during product derivation as well, and some automatic guidance and assistance might thus be needed during the process of product derivation [RGD10]. Advanced feature diagram visualization techniques that aid with product configuration can be found in a study by Pleuss et al [PB12] for example.

Finally, there are some basic functional features that affect usability, but are not necessarily required for achieving any goals the user has. These include feedback to user, ability to undo actions, input validation, UI shortcuts, UI customization, cloning and other reuse of data, and user help [EDDT11]. Related to feedback, reporting and explaining anomalies such as dead features in an understandable way and accurately can help with guiding a user through the product derivation process [KAT16].

2.2.2 Other factors affecting tool choice

One important aspect to consider when selecting a tool is whether and how well a variability modeling tool integrates with other tools used in software engineering, and in general how easy is it to incorporate the tool in different software processes [CSD07]. The tool should also be easy and light-weight to install and maintain, and migrating to and from it should be easy as well [DSF07]. Some other, related fac-

tors include availabilty of example solutions, documentation, source code, customer support, a trial version of the tool itself, and source code [EDDT11][PCF15]. The activity of ongoing development and maintenance of the tool or the lack of thereof is an aspect worth considering as well[PCF15].

Some other important aspects to consider are usability, learnability and user experience. Different tools can come with very different user interfaces for achieving same goals. For example, different approaches to input of information are discussed in [EDDT11]. Since the tools do not necessarily follow the exactly same variability modeling techniques by the book, they can have differences in their visual syntax as well. These differences can affect usability [SSAIEA16]. In addition, the textual syntax used by the tools can vary in terms of how easy it is to learn or reason about, for example. Furthermore, in some variability modeling techniques and the tools implementing them, variability modeling is done completely textually. The benefits of this approach such as scalability, power and ease of integration into the general development toolchain as described in [ES13] need to be weighed against the benefits of the graphical approach, including the relative ease of getting started with a graphical notation.

The application domain and size of the organization can be factors as well. The types and skillsets of stakeholders can vary, and some domains, such as industrial automation, can have their own complexities that require special support from tools [FLD+15]. In the systems software domain, constraints seem to be used especially extensively when compared to other domains, for example. The degree of completeness of the feature models can thus vary by domain, and the tool's strategy for automatic analysis and configuration among other things can be more or less optimal for a given domain. In some domains, highly domain-specific languages and tools are often used for SPLE. When it comes to organization size, larger ones are probably more likely to require support for reuse and sharing over multiple product lines, and more sophisticated support for a larger number of users. For specific requirements for multi-PL support, the reader is advised to see the study by Holl et al [HGR12]. On the other hand, small and medium-sized organizations tend to favor agile methods and generally a more light-weight approach than larger ones [BE13b].

2.2.3 Pure::Variants

Pure::Variants (PV) [PV04] is a commercial tool for SPLE. It supports a wide array of SPLE activities and is one of the most popular if not the most popular single

SPL tool used in industry. It can be installed as a plugin to the popular, open-source IDE Eclipse. A server style and an integrated deployment options are also available. PV supports linking the models to code, enabling producing software products in a fashion resebling the mode-driven development (MDD) paradigm. PV integrates readily with the popular requirements engineering and management software DOORS, and with Rhapsody, an MDD tool for embedded systems.

Pure::Variants uses similar feature modeling concepts as CBFM, with a few differences. First, there is support for mapping parts of the FM to a single feature in another FM. This results in multiple, hierarchical feature models with varying levels of detail. This can help with scalability and suitability for different stakeholders. Second, there is no support for references, and the scalability is dealt with in other ways.

One of the distinguishing features of PV is the *family model*, which relates the features of the FM to the actual software components of the SPL. The family model is a hierarchical tree-like model, where components consist of parts that can have many pieces of sources. The components implement one or more features from the feature model, and the sources are actual source code elements.

2.2.4 Clafer Tools

Clafer Tools (CT) [ABM+13][BDA+14] is a collection of interoperable, non-commercial tools supporting feature modeling with cardinalities and product configuration based on the models. CT supports the typical SPLE process described earlier at least on a basic level, and a lot of focus is given to scalability as well. All of the tools are open-source, and binary distributions for them as well as an online demo instance are also publicly available.

Clafer comes from the words class, feature and reference. The Clafer language aims at a minimal number of concepts while still not sacrificing expressiveness. It does this through unification of several concepts. These concepts are class, feature, instance, reference, attribute, and feature group, all of which fall under the concept of a clafer. The Clafer tools use the Clafer language for modeling, and the models are built and managed textually. One more interesting characteristic of Clafer tools is the support for user-defined optimization objectives and generation of optimized configurations based on them. The objectives are based on information about non-functional properties and other additional information included in the

extended model.

Clafer tools does not include any tool support for linking models to pieces of source code or configuration files. The product configurations only serve as means of communication and guides for software engineers who have to handle the variability on the level of source code and configuration files. No mentions of any integrations to third party tools that could provide this functionality was found either when searching through sources for this work. However, in principle the textual models should be relatively easily importable and usable in other tools, although this again falls out of scope of this master's thesis.

2.2.5 Evaluations of the tools

Berger et al [BRN+13] present an industry survey of SPL practices, approaches and tools. Pure::Variants was found to be by far the most widely used single tool in the industry, with 35% of the participants having used it, followed by Big Lever GEARS with 23.5%. However, the number of reported tools was nearly as large as the number of respondents, and additionally as many as 38.2% had used a home-grown domain-specific tool. These results suggest both that adequate support for SPL by existing tools has been lacking, and that the field is very fragmented, which further contributes to difficulty of selecting a tool.

Pereira et al [PCF15] reviewed a large number of SPLE tools on the basis of which functionalities they provide any support for out of a pre-selected set of functions, based on research papers. The authors found most tools lacking in user guides and example solutions, and many of the tools themselves being complex or unavailable. The list of properties and actions for which the support provided by the tools was reviewed can serve as a good basis for further, more in-depth evaluations. Since the results show that for every property or action there is a fairly large percentage of tools that provide support for it, it can be concluded that the body of available research in SPLE methods and tools generally agrees that the evaluated functions and properties need to be supported by an SPLE tool.

Djebbi et al [DSF07] evaluated SPL tools based on a number of criteria categorized in product line engineering criteria, management criteria and technical criteria. The list of evaluated tools included four tools that had good enough availability, seemingly sufficient support for industry use, and were found practical enough after a small use case study. These tools were Pure::Variants, XFeature, RequiLine, and DOORS

TREK. PV and RequiLine were found to best match the set criteria. Constantino et al [CPP+16] compared Pure::Variants and another Eclipse-based tool, FeatureIDE based on feedback given by a group of students using each tool respectively. The students gave feedback on how hard or easy it was to perform FM edition, automated FM analysis, product configuration and FM import and export with the tool they were selected to use. El Dammagh and De Troyer [EDDT11] evaluated nine SPLE tools - including Pure::Variants - based on three sets of software quality criteria: usability, safety and functional usability features.

3 Research methods

In this thesis, two different experimental studies are performed. The first goes through requirements for SPLE tools obtained from literature with the aim of finding out how well each tool supports them. The second study is a report of experiences from using both tools to create a part of a fictional SPL.

3.1 Methods of the first study

3.1.1 Context

The context where the tools are used here is assumed to be generic, i.e. no assumptions about the application domain or organization are made. However, support for extractive SPL development is not considered, and generating code assets directly from the tools is left out completely due to Clafer tools not providing any support for that, and due to scope limitations of a master's thesis. So it could be said that the context is one where modeling is emphasized or where an integration between modeling and code generation is not necessary.

3.1.2 The assessed functionality

As was implied in the literature review, there exists no standard for what should be required from an SPL tool. Nevertheless the literature and the available tools seem to largely be in consensus regarding a set of basic functionality when it comes to feature modeling for SPLs. The evaluated basic functionality are adding and editing features, constraints, attributes, model validation and error checking [CSD07][CPP+16][PCF15], inheritance and modularization techniques [CGR+12] [BRN+13][CHE04], and creating and editing configurations.

Since extractive SPL development is left out of scope of this study, features need to be added to the models manually. Additionally, real-world feature models can contain large amounts of features, and adding and managing them can consume significant amounts of time. Thus adding and editing features can be considered as some of the most important basic functionality of SPL tools when it comes to feature modeling.

Along with features, constraints and attributes [PRB11][SD07] are other main elements of which feature models used for SPL consist. Constraints are included

in FODA and its extensions [CHE04], and are very widely supported by tools. Attributes are supported by both studied tools and often by other tools as well [PCF15]. Different projects may have different needs and strategies when it comes to constraints and attributes [BNR+14][BRN+13], so looking at each of features, constraints and attributes separately may be useful.

Application engineering constitutes a very significant part of SPL engineering activities. However, since this study focuses on modeling and leaves product derivation out of its scope, product configuration tasks are included in the evaluation only as a single category.

3.1.3 Evaluation method

For each piece of functionality or non-functional property mentioned in the previous section, the level of support provided by the tools for it will be assessed, and then scored from zero to three. Zero is no support at all, while other scores are based on context coverage [IEC], efficiency, functional suitability and usability of the tool when it comes to the functionality or property being assessed.

Cardinality-based feature modeling (CBFM) [CHE04] is especially used as a reference point when considering what should be expected, as it is very frequently talked about in literature [EKS13][SD07][ABMG12] and many feature modeling tools have the equivalent functionality as in the definition of CBFM. Other specific requirements used in this study are automatic configuration generation and visualization of the configurations, partial configurations, default features, detection of dead features, validity checking, and helpful error messages. All of these are discussed in the literature review and the studies it refers to.

3.1.4 Limitations

The application domain and the type and size of the organization affect the specific requirements for an SPLE tool, so a generic evaluation such as this might not always be useful for practicioners. In addition, the set of functionality and requirements is somewhat arbitrary and not standardized.

3.2 Methods of the second study

3.2.1 Study context

Since the suitability of a given tool for SPLE tasks is so dependent on the context, trying to make generalizations for a tool's suitability for a generic SPL might not be feasible or produce any useful results. Moreover, there are domain-specific tools and languages for some domains, such as operating systems, that are probably better suited for their intended domains than any of the more generic tools. Therefore, it makes sense to select a specific context for applying the tools and perform the comparison and evaluation for that context.

The example context where the SPLE tools are applied in this study is an imaginary web-based board game and card game platform that is built with the SPLE approach. A digital board game platform is a very natural fit for SPLE, as almost all of the games share a similar turn-based interaction model, and game mechanics and component types are widely reused across the industry. Secondly, there seem to exist no domain specific tools for this domain, so applying more generic SPLE tools makes sense. Thirdly, there appears to be little if any research on applying SPLE on this domain regardless of the high applicability. Thus board games being chosen as a domain for applying SPLE in could be particularly fruitful for inspiring future research. Lastly, not as much deep or specialized domain knowledge is likely to be needed to create a relatively realistic feature model and to understand it, than in domains such as industrial automation or operating systems.

The organization developing the platform is assumed to be small and just starting out and experimenting with SPLE, and the approach used is mostly reactive. Additionally, a generally light-weight development process is assumed, meaning that formal requirements documentation and importing the requirements into the tools will be skipped.

3.2.2 Test method

A small feature model will be developed and made separately with both tools. The models in different tools will be kept equivalent to the extent that this is possible. Any findings that are relevant for the studied software quality characteristics (see previous subsections) will be reported on and categorized as strengths or weaknesses. The experimental study is divided into three parts: Installation and setting up,

building a feature model, and product configuration.

3.2.3 Evaluation and analysis

The found strengths and weaknesses are finally summarized, and subjective analysis is conducted to see if any recommendations or suggestions for improvement can be made based on the results.

3.2.4 Limitations

The work will focus mostly on tasks related to modeling due to both time limitations and the fact that both tools appear to offer comparable support only there. Support for testing, integrations, requirements engineering and automatic building of software is left out of the experimental study, and will be only discussed as a part of the literature review.

The approach taken in this study is mostly exploratory - what is being done is not thoroughly planned beforehand. This is because it would be hard to predict ahead of time which tool functionality and properties will be important, and to which extent, due to the difference in needs for different contexts. While it would be possible to create models of the example product line in some other tool first, and then list required or important functionality and assessment criteria for a tool based on that experience, that approach would have its own drawbacks. The most obvious of the drawbacks is the scope of the study getting out of hand. Another is that such an approach would not be a good fit for the chosen context, where the emphasis is on experimenting and prototyping with SPLE. Finally, tools come with their own collections of SPLE methods and approaches, and required tool features and characteristics derived from a model made with a third tool might not be well applicable to the studied tools due to differing modeling techniques and methods.

Since there are no easily applicable objective measures available for the studied quality characteristics, the study will be more or less subjective, and the results not well generalizable or easily reproducible. There was no easy access to users of either tools, and especially people who have done comparable work with both tools would have been hard to come by.

3.2.5 About the example system

As said in the previous section, the example for a product line model used here is a digital board gaming platform featuring many different games. As explained earlier, there are no formal requirement specifications for the product line or individual products, nor is there a reference model of a product line that could be copied to the tools. However, an informal tree diagram (figure 1) was made of the potential features in the product line. As the emphasis in this study is not on planning and requirements engineering, the diagram should not be considered final or complete. It exists mainly to avoid the problem of having to define all the features and their relationships on the fly while creating the model, which could damage the validity of the experiment and the whole study. In addition, it should help the reader understand what is being done and why in each step.

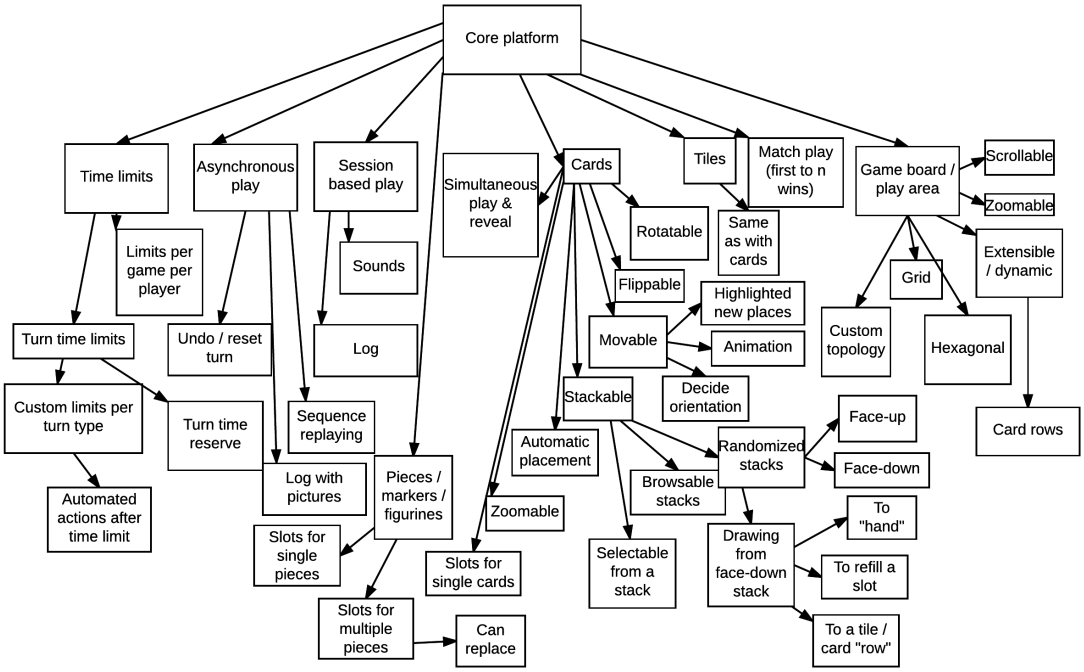

Figure 1: Sketch of a feature tree for a board game SPL

4 Results

4.1 First study: support for requirements found in literature

4.1.1 Feature addition

Pure::Variants

Adding a feature is fairly easy and straight-forward to do in PV. See figure 2 for the feature addition pop-up dialogue. There are two name fields, one functioning as an internal unique name, which is mandatory, and another meant as a nice, human-readable name, which is optional. The usability isn't as good as it could be here, as the unique name does not get generated automatically based on the human-readable name, and if a nice name is wanted, both names have to be typed manually and separately. The user interface sometimes lags slightly, and sometimes it doesn't commit actions on the first try.

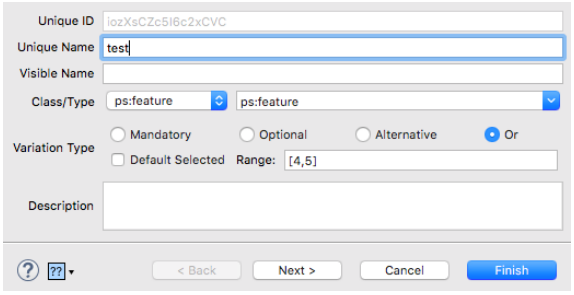

Figure 2: Feature addition dialogue in PV

The following sequence needs to be done when adding a feature. First, the intended parent feature is clicked with right mouse button, and 'add feature' is selected from the pop-up menu. Then the unique name must be filled, and inclusion criteria selected with the mouse. Then the button that closes the menu and actually adds the feature needs to be clicked. There may be an alternative path for entering the menu, but no keyboard shortcuts help with this according to the manual.

The feature addition dialogue supports all the CBFM concepts. There is the possibility to create a group of sibling features, from which n out of m features have to be selected. Unlike in CBFM, there is support for default features, i.e. features can be made selected by default. In PV, all sibling features with the 'or' type are forced into a single cardinality group. This could be standard practice, but it is not at all deducable from the user interface for someone not so familiar with feature

modelling. While it is probably not common to have need for multiple cardinality groups under the same parent feature, the implementation of PV leads to unexpected behavior where minimum and maximum numbers of features defined earlier for a group are replaced by completely different numbers without notice, when the user simply intended to create a new group.

The feature model is validated automatically whenever there are changes. Additionally, changes become instantly available in the product configuration tab, so the user can easily verify that the model works as intended after trying out something he/she is unsure of how it works. This is very useful for learning.

Clafer Tools

In CT, features are added purely textually, by putting the desired name of the new feature on its own row in a text file. The nesting of features in the tree structure is decided by indentation. Operators can be added both before and after the feature name for defining the criteria and logic for their inclusion and their feature- and group cardinalities, among other things.

CT supports all CBFM concepts. Features are mandatory by default, and optional when followed by '?'. A group of alternative features can be defined by preceding the parent feature with 'xor'. Other types of group cardinality are defined similarly. A feature group from which at least n and at most m features must be selected, and where n and m are arbitrary, can be defined by preceding the group's parent feature with n..m. Additionally, feature cardinality is supported, meaning that it can be defined that at least n and at most m copies of a single feature is included in a product that includes the parent feature.

The syntax is quite concise, and in general the clafer language seems to be designed for both high productivity and flexibility. Unfortunately the same design choices can reduce learnability. For example the flexibility enabled by the cardinality definition for both features and feature groups leads to there being multiple ways to express the exact same things. It can be confusing for new users to see feature cardinality of 0..1 sometimes used instead of '?', or 1..1 being used instead of the keyword xor.

CT has no support for default features at the time of writing this, although they are apparently planned.

4.1.2 Constraints

Pure::Variants

In PV, there are actually three different concepts that resemble constraints: relations, restrictions and constraints. Relations in PV can be defined through the GUI and are functionally similar to cross-tree constraints from FODA, with some differences. One difference is that they are always nested under a single feature. Another is that there is no way to express a constraint that two features cannot be selected at the same time through a single relation, and to achieve the this, relations have to be defined for each feature separately. Thirdly, mutual exclusion (xor) and both-or-neither (and) relationships between two features require adding two relations, although they can both be added for only one of the features. Conversely, there are some relation types in PV that are in a sense more powerful than basic cross-tree constraints. The relation *ps:conditionalRequires* can be used to declare that for this feature to be selectable, the other feature has to be selected too, but only when its parent is selected. This cannot be expressed through a basic cross-tree constraint. With just typical logical expressions, the parent feature would need to be explicitly referred to, and the constraint would be much more complex, easier to get wrong and less readable. Similarly, a mutual exclusion for more than 2 features can be done more simply and cleanly by giving all the features the relation *ps:exclusiveProvider(id)* with the same id. There are also "softer" versions of many of the relations, that generate warnings instead of errors, when not true.

Constraints in PV are a stronger and more flexible version of the relations, with the limitation that they cannot be defined through a GUI. There are two different textual logic programming languages the constraints can be written with: pvProlog and pvSCL. pvProlog is a dialect of the more widely used Prolog language, which has the advantage of being used elsewhere as well and therefore more people being familiar with it, while pvSCL can be more compact [PVMAN]. Restrictions can be thought of as constraints that are specific to a single feature. They are written with pvProlog or pvSCL similarly to the PV constraints. PV restrictions are a concept similar to derived features, with the difference that even when a restriction does not disable a feature, it still might not be included in a configuration. While the same thing could be achieved through constraints, restrictions are simpler to define and easier to understand in some cases.

One difficulty with using constraints, relations or restrictions is that they need to

refer to the unique names. The unique names are hard to remember, and are not easily seen at a glance from either the tree or graph views. There is a table view that shows them for all features however. In addition, when typing the first characters of the unique name, the UI should give a list of features that start with those characters. It might be hard to come up with a naming scheme for the unique names where it's both easy to remember the first characters and the names aren't cumbersome to type when creating the features. Depending on how heavily relations, restrictions and constraints are used, either brevity or obviousness of the names can be emphasised in the naming scheme.

Clafer Tools

CT supports propositional logic costraints as in FODA. Additionally, conditional expressions are allowed in constraints as well. The constraints are written simply by enclosing names of features and logical and conditional operators in square brackets. Similarly as in programming languages, the name of one feature can be replaced by the keyword *this* that refers to a feature the constraint is nested under, i.e. the context. This can be helpful for readability. Constraints can be defined globally as well by simply putting them at the beginning of a line, without indentation.

4.1.3 Attributes

Pure::Variants

PV allows adding attributes to features while creating them or by adding them later. You can choose one of the predefined attribute classes or create one of your own. The predefined attribute classes, such as risk, have a predefined set of values, such as 'low' etc. Attribute classes created by the user can be reused just like predefined attribute classes. Attributes are nested under the features they belong to in the tree view, and values of the attributes are nested under the attributes themselves.

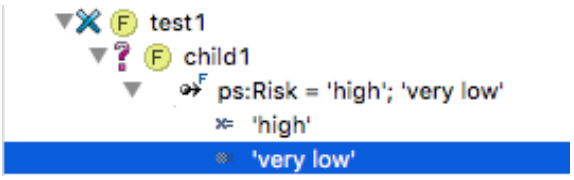

Figure 3: Attribute with two values

An attribute can be given multiple different values (see figure 3), but the user interface gives a warning in that case that only the first value is ever used, unless *restrictions* for the values will be defined. Restrictions are similar to constraints, and with them one can define conditions for a given attribute value to be applied. The restrictions are defined with a textual language, which allows flexibility with the conditions. The drawback is that the user will have to learn one of the languages that the restrictions can be defined with to control attribute values in this way. It could be argued that this partly defeats the purpose of having a graphical variability model editor in the first place.

Clafer Tools

In CT attributes can be thought of as features that are at the same time variables of some type. A feature is turned into an attributed feature by adding a semicolon followed by the attribute type (e.g. integer) after the feature name. Feature cardinality is added just like with non-attributed features by adding '?' or some other feature cardinality at the end. For example, an optional feature describing a game ending condition triggered by a player reaching a certain amount of points could be defined like this:

gameEndPoints : integer ?

Restrictions on attribute values - including setting them as constant - can be imposed through constraints. The same syntax as in cross-tree-constraints meant for deciding feature inclusion is used, with just the difference that any operators suitable for the attribute type can be used instead of just logical operators. So attributes are like features in this respect as well. Overall, attributes in CT seem to be a simple and straight-forward implementation of CBFM's attributes.

4.1.4 Model validation and error checking

Pure::Variants

Being an Eclipse plugin, PV can make use of the automatic build functionality in Eclipse. When that is on, PV is constantly checking the validity of the model and building the configuration space based on it. However, the validity and error checking for the model is mostly limited to syntactic correctness. When the validity check was tested, no errors or warnings were shown for the family model even when

there were two features that were mandatory in every product, having relations that would be impossible to satisfy at the same time - one of the features had a 'conflicts' relation targeting the other, and the other one had a 'requires' relation targeting the first. Such a model is clearly broken and useless as it can produce no valid configurations, and therefore would definitely warrant an error message or at least a warning shown to its builder. The only thing generating a warning for the family (feature) model that was observed was a feature having multiple values for a single attribute without restrictions for the values, and therefore the first value always being used by default.

To find out errors having to do anything with product configurations, such as the relations described above, one has to look into the configuration editor, even though the error in many cases should be easily detectable from the family model and exists because of a mistake in its definition. Furthermore, the errors that are pointed out in the configuration editor only point out specific relations that do not hold, instead of reporting the logical conflict, which should be relatively easy to detect automatically. A related issue is that dead features, i.e. features that are never selectable in a valid model, are not reported on in any way, even though they are obviously mistakes in model design.

The automatic validation and error checking can cause noticeable lag to the user interface when editing the feature model or configurations. However, the validation can be configured to be done only manually, or the scope of the automatic validation can be decreased to reduce the computational load.

Clafer Tools

When using the SublimeText plugin for CT, the clafer model can be built with the keyboard command for building. The build process does not generate any errors or warnings unless there are actual syntax errors. Dead features, impossible constraints or anything like that do not generate errors or even warnings. The interesting things about the output, barring syntax errors, is a short summary of the number and types of clafers generated, and the number of constraints, although even this info is of limited use. The build time is shown as well, which is at least interesting for research purposes. The build takes several seconds even with a very small model of only three features and a single constraint on the tested system.

Any design mistakes can only be detected in *instance generation*. The instance

generators basically generate product configurations automatically, and they can be invoked through keyboard commands. The instance generator does detect constraints that are in a mutual conflict, telling that they "cannot be satisfied in the current scope", and pointing out the names of the constraints and the rows they are defined on. Dead features generate no warnings. It should be noted that only the Alloy-based instance generator, ClaferIG, was used here, as the Choco3-based instance generator does nothing but report an unhandled compilation error on the used setup. The third option does not work either, but the support for it seems to be more limited than for the other two in any case.

4.1.5 Inheritance and modularization techniques

Pure::Variants

Product configuration in PV is done through *variability description models* (VDM). A VDM represents a partially or fully configured branch of the feature tree, and it can be reused or inherited by other VDMs.

While testing how Inheritance of VDMs exactly works, the following problem was discovered. The PV community edition does not allow opening multiple VDMs at the same time, which would not be a problem by itself, but apparently the system internally considers inherited VDMS opened at the same time as the inheriting VDM is opened. Inheritance is not mentioned anywhere to be among the things missing from the community edition, and the error message when attempting to use it is not providing any hints as to this being the case. Therefore, it can be concluded that this is indeed a bug. According the the PV user manual, VDMs can inherit single or even multiple other VDMs [PVMAN], but it is not entirely clear how this works on a detailed level. In any case, this functionality should provide a way to make partial configurations and inherit them for other configurations, enabling selection reuse.

VDMS and collections of them that are from a different feature model can be linked as sub-trees into a feature model according to the manual [PVMAN], but this does not work either in the tested setup for the same reason that inheritance does not. When it works properly, this functionality should enable reusing feature models as parts of other feature models, and modularization of development of the SPL.

Clafer Tools

CT supports inheritance of abstract clafers, which can represent branches of a feature model tree. As inheritance in CT works very similarly to inheritance of classes in object-oriented programming languages, the benefits are largely similar as well: enabling reuse of parts of models and modularization of code and improving comprehensibility. Partial configurations are made possible in this way as well. Another functionality related to inheritance are *references*. Features can refer to other features with an arrow syntax. The difference between instantiating an abstract clafer and referring to one is that the former creates an independent instance, which inherits from the abstract clafer, while the latter refers to any instances created from the abstract clafer.

Inheritance is very useful in large models for the reasons explained above. References could be useful in a situation where there are multiple different implementations of some feature or a branch of the feature tree. They can be maintained separately, and they will be put in place of the referrals in automatic configuration generation.

Both inheritance and references in CT suffer from the problem that both the abstract clafers and their instances need to be at the root level of the same file to be available for instantiation or reference. This can make model files unnecessarily cluttered and large. In addition, the instances that can be used by references cannot be abstract, and are always included in generated configurations at the root level, adding unnecessary information to the configurations.

4.1.6 Creating and editing configurations

Pure::Variants

Inclusion of features in the VDM can be changed by checking or unchecking checkboxes shown before feature names in the list of features available for the VDM. Attributes can be changed similarly to fields in a form by clicking to make them editable and then typing the new value.

When the inclusion of any feature or the value of an attribute is changed, the validity of the change is checked, and errors and warnings generated as mentioned in the previous section. The inclusion of the features' child features or some other features the added or removed feature affects through relations, restrictions or constraints can change dynamically. Such automatic changes can be hard to notice, and there is

no highlighting of them. Moreover, relations, restrictions and constraints can make features not selectable or not deselectable, and in that case attempting to change the selection results in the change being reverted, without any indication of why that happens.

With PV there is no built-in support for viewing and browsing potential configurations. Generating XML files that correspond to the configurations is supported, but they are not useful for human readers, and are meant for generating code assets. There is a javascript API that also generates machine readable product information, and using it to generate useful, human readable information would take some coding effort at best.

Clafer Tools

The SublimeText plugin is quite limited when it comes to creating configurations, when using the Alloy instance generator. Again, the Alloy generator was the only one that would work with the tested setup. Additionally, there is no way to edit configurations.

When the instance generator is called, the plugin performs a validity check, and then opens a command line interface for the generator to another tab of the editor. The first instance is generated and shown automatically, the rest will be printed one at a time when hitting 'n'. The instances are numbered, and seem to be printed in reverse alphabetical order. The only customization or configuration options are setting the *scope*, and choosing whether to use a faster but worse or slower but better version of the satisfiability solver. The scope determines the maximum number of times a single clafer can be included in a configuration, and defaults to 1. It is only interesting when using cardinalities larger than 1 for features (or more generally clafers).

The Clafer Configurator (CC) can be used for both generating and editing configurations based on a Clafer model. The configuration UI is a table where rows represent features, attributes and constraints in the model, and columns represent variants that exist for the given configuration choices. CC supports partial configuration and automatic configuration generation based on the partial configurations: optional features can be both included and excluded by clicking checkboxes after their names, and the list of matching variants updates immediately. CC supports searching or filtering of the list of clafers by name simply by typing a part of a name

into a field above the list of clafers. This is very quick and convenient for the user. Additionally, nested clafers (e.g. sub-features) can be hidden and made visible again by clicking arrow symbols next to the names of clafers that have child clafers.

One problem of CC is that it does not integrate well with SublimeText plugin. If you have compiled a model and generated instances based on it through the plugin (or Clafer IDE), you will still have to either open the model in a non-compiled form, i.e. as a plain-text file in CC or copy-paste it into CC's own editor view, and generate instances again there. The editor view of CC can be expanded, and it has syntax highlighting, row numbers and collapsing and expanding the text based on nesting, but there is no option to save the model to a text file. Thus it would be inconvenient to use for editing the model.

4.2 Second study: suitability of the tools for a board game SPL

4.2.1 Installing and maintaining the tools

Both tools come with different options for installation. Only a single installation option will be used for both tools in order to not let the scope of the study get out of hand. Simpler and easier methods of installation and usage are preferred in this study, as the viewpoint is that of a small organisation hoping to avoid investing a lot of time or money into setting up and managing the tools. Both tools are installed locally for a single user on a Macbook Air running OS X version 10.11.6 (El Capitan). They are also tested on version 10.13.1 (High Sierra).

Pure::Variants

Pure::variants (PV hereafter) is commercial software, but an evaluation version is freely available, and it is that version that is used here. The evaluation version is adequate for the needs of this study, as it's mostly just the integrations that benefit from the full version. The evaluation version is installed as a plugin for Eclipse on a single desktop, while the full version can readily be deployed in a distributed manner for multiple users.

The Eclipse version used with PV here is 4.6.3 (Neon). The officially supported Eclipse versions for PV are the ones from 3.6.0 to 4.3.x. However, the newest version of Eclipse is 4.6.3 as of May 2017, and versions older than that are no longer supported. A choice had to be made between going with an older version of Eclipse that is officially supported by PV, but that is deprecated and thus not updated for a long time, and going with the latest version of Eclipse that is not officially supported by PV. The latter option was chosen, because it would be a big drawback of PV if it was dependent on a legacy version of Eclipse. Using an old version of an IDE for all development work might be out of the question, and using a separate development environment for the actual development work would sacrifice any benefits of the tool being included into an IDE as a plugin. On the other hand, there seems to be little information available on whether PV works on the newest Eclipse version, and the only way to find out seems to be to test it. If there are problems because of this, this can be reported, and in that case it can be said as a fact that PV depends on a legacy version of Eclipse in order to work correctly. If the problems are severe

enough, a switch could be made to the older Eclipse version.

PV has been developed and maintained quite actively, with no more than a few months between new, minor versions. The last major version, 4.0.0 was released in April of 2016. From looking at the version change log, it appears that the changes from 3.x.x are not breaking and consist of usability improvements, new features, and bug fixes. Additionally, there are many changes to integrations with other tools such as enabling custom paths for exporting DOORS modules, and support for Office 2016.

Installing PV in Eclipse is as easy as entering the address of the PV update site into Eclipse and clicking a few buttons, and fetching updates for it once installed requires only a few clicks. From the ease of updating and the observation that the software additionally seems to be fairly mature and does not introduce breaking changes often if ever, we can conclude that updates should pose no problems for using PV and maintaining an installation of it. As there is a fairly high rate of changes to integrations with other tools, the situation might be a bit different when there is a high level of dependence on them. However, this study does not go into details about the integrations.

Several new versions of PV were released while this thesis was worked on, and by November 2017, the newest one is 4.0.9. This version works very poorly on the tested machine. For example, it made product configuration impossible to access, as the PV process would start hogging up 99 percent of CPU cycles and get stuck indefinitely. This behaviour did not change with newer Eclipse version (Oxygen), with different version of OSX, or with different Java SDK versions. To make matters even worse, older versions of PV are not accessible, at least for customers of the evaluation version, so we are stuck with a non-functional setup.

Clafer tools

Clafer tools (CT hereafter) can be used in two different primary ways: as a multi-user server-based setup, where a wiki is used as an IDE, and as a single desktop setup, where the tools are integrated into a package for Sublime Text version 2 or 3. Both can be built from sources, while there are binaries available for Linux, Windows and OS/X for the latter only. CT's website recommends the latter setup for desktop usage to get the best user experience, and it is the one chosen for this study.

New releases of the tools have been made available every 3 months on average for the past four years, with no more than half a year between versions. The first version of the current (as of May 2017) 0.4.2.x branch, 0.4.2.1 was released in November of 2015, and introduced some new syntax, new features and fairly significant improvements, but no breaking changes. The last major update, 0.4.0 was released in July of 2015, including some changes to the grammar and type system, but apparently nothing that would have broken backwards compatibility significantly or extensively. So as with PV, it can be said that the software is fairly mature, and updates will probably not cause much issues, and that the software is actively maintained and even improved.

Installing the tools as a integrated setup with Sublime Text requires two simple steps. First, the Clafer package must be installed from Sublime Text package control. Next, a release of Clafer tools needs to be installed and its command line interface made available for the Sublime Text package to use. Both steps include some manual moving of files. Updating consists of similar steps as installing, and both can be done in a few minutes once the user is familiar with the process. However, installation to certain systems requires some additional tinkering or workarounds. These are described for some common operating systems in the installation guide of the integrated setup. There were no problems with installing on OS X version 10.11.6, but it's possible that there are some operating environments that require different workarounds to be able to use the software.

To get instance generation to work conveniently from SublimeText, it was necessary to locate the keybindings coming with the SublimeText plugin and change them to something else in this setup. This is likely to be a problem related to the operating system the tools were installed on.

Although the documentation recommends only the Sublime Text setup for "the best desktop experience", it turns out ClaferConfigurator was needed as well to have acceptable support for product configuration. Luckily it was fairly easy to install and works well out of the box without any extra configuration steps.

4.2.2 Building the feature model

Here a model of the example system is built with both tools. The scope is limited to only the 'Cards' branch of the tree (see picture from earlier) due to PV starting to malfunction while it was being made. Pictures of the models can be found in the

appendix.

PV

Adding a large amount of features quickly gets cumbersome in PV and takes a lot of time due to the large amount of pointing and clicking required. The user interface often lags a bit, and sometimes very badly, making the experience even worse and increasing the time it takes to add many features. This was definitely a problem when adding the features for the Cards module, as the list of features and their relationships was already mostly known. The task could have been completed in much shorter time if the UI would have allowed it. The lag in the UI probably partly results from automatic validation, but it was not possible to make it disappear or negligibly small on the tested setup. The positive thing about the feature addition dialogue is that it isn't any more complex than it needs to be, and the amount of options is kept to a minimum. This makes it easy for beginners.

The human readable names are useful, and greatly improve the readability of the model. In addition, they do not need to be unique, so they also make it possible to refer to same exact functionality in many places by using the same human-readable name in all of them. In some situations this could make more sense than adding actual references to features defined elsewhere due to it being more simple. In the model the 'Auto place' and 'Selectable place' feature names are used in two places even though they are separate features with different unique names. The free-text descriptions for features would probably be useful for anyone trying to understand the model, when there is no external documentation of the feature model available. In our case we assume a very small, co-located team, so the benefits from creating and managing the descriptions might not be worth the time it takes.

Adding attributes and constraints is much more cumbersome and harder than adding features. The user interface for adding or editing an attribute or constraint functions quite poorly on the tested system. Usually fields need to be clicked many times before they can be edited, and the fields are very small. Additionally the user is presented with unnecessary and potentially confusing information like internal attributes such as 'source'. Furthermore, PV seems to make the assumption that the user building or managing the feature model wants to add multiple values for attributes and constraints into the feature model. This is not always true, like in our case just an acceptable range was defined for attributes (varHandLimMax and varHandLimMin). Finally, only exact values for attributes and constraints can be

given easily from the user interface. For everything else, the user is given a big editor window, where code for determining the values must be written with pvProlog or pvSCL. This is quite slow and cumbersome for expressing many simple things such as maximum and minimum values for attributes, like in our model. In addition, it adds significantly to the amount of learning required to use PV efficiently.

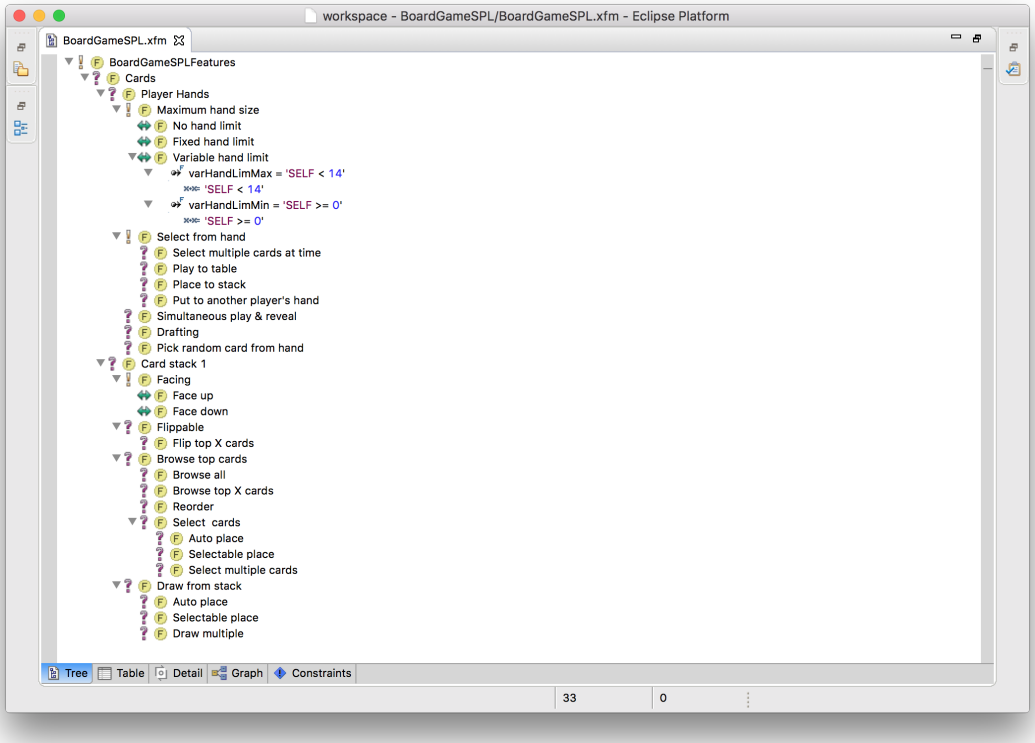

Figure 4: A board game feature tree

When it comes to features in the feature tree, the visual syntax is quite intuitive and easy to understand (see figure 4). If the structure of the model is not easy enough to see from the tree view, there is also the graph view (figure 5) that emphasizes it. Features can be added or edited and any other modifications of the tree can be made directly from the graph view as well. It quickly becomes obvious that it doesn't scale as well as the tree view, but the ability to hide children of any element by clicking its upper right corner helps with the scalability. Nevertheless, folding and unfolding elements is additional work, and the graph might be better suited for just visualization. Additionally, elements are placed in the graph without any intelligence (e.g. on top of each other) and need to be manually dragged to places that make more sense. Attributes and constraints are presented in the tree view in

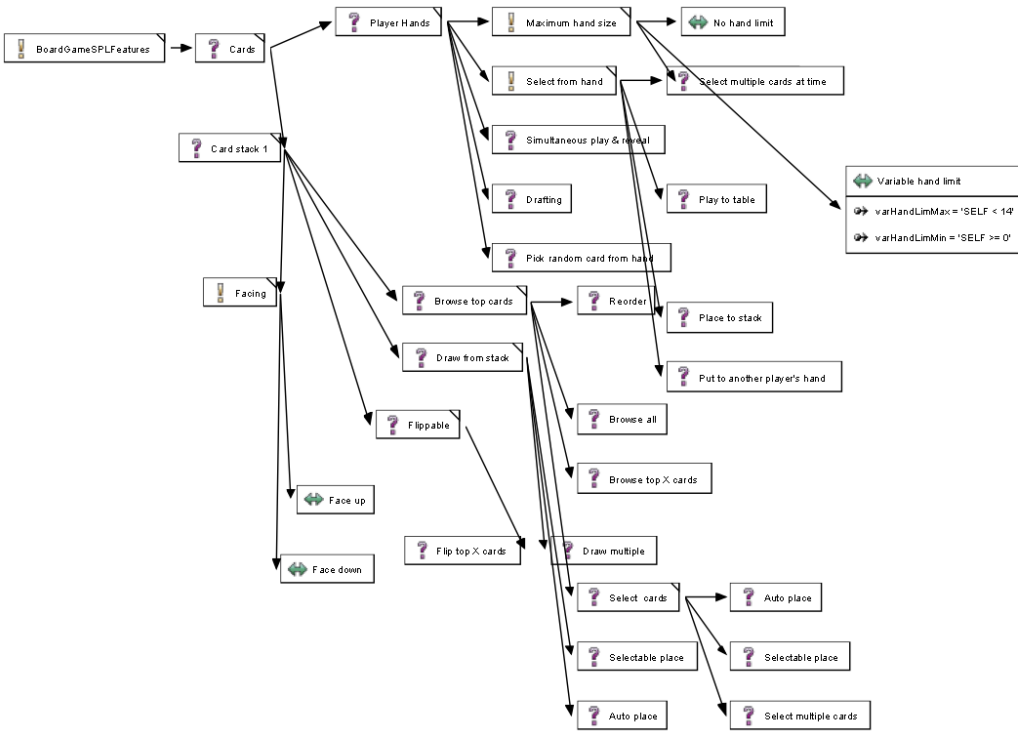

Figure 5: An FD graph generated by PV and then hand-tuned

a way that is not optimal for our model, as there are only single values that get repeated on another row pointlessly. The way they are represented in the graph view makes much more sense for our model.

CT

The Clafer model is quick to write and edit in SublimeText, thanks to the simple and concise syntax. There naturally are no more wait times or lag either while editing the model than what would be experienced while editing any text file on SublimeText, since there is no on-the-fly checking of model validity or even syntax of the text file. The syntax and basic validity is checked when the model is compiled, which can be easily done with a keyboard shortcut. The errors caught are what could be expected of resulting in compiler errors with programming languages, like trying to instantiate a nonexistent abstract clafer. The compiler errors that are shown in the editor window are easy to understand and thus to locate as well. The partial model having been done here (see figure 6) took 0.7 seconds to compile on the tested system.

There are multiple ways to present most ideas in CT, as the language is quite

flexible but also rather abstract or high level. Building a feature model without any abstract or reference clafers would be quite straight-forward when one is familiar with the syntax, but even from just the Cards branch of the feature model, we can see that it would be impractical. One obvious drawback would be complexity and manageability of the model code. Perhaps a bigger problem would be inability to create different types of card piles for example without repeating a lot of code. However, coming up with the best or even reasonably good ways to modularize the model with abstract and reference clafers is not simple. The developers of CT have a wiki that includes many examples that help, but many of them are closer to domain models or models of concepts than feature models applicable for SPLE. For example, there are example solutions for a book, and a railway network. Such examples' usefulness is limited apart from showcasing the syntax. Furthermore, there exists alternative syntax for many things, and there have also been some changes to the syntax since some of the examples and some other material in the wiki has been written, which reduces the wiki's usefulness as a learning material. Finally, it can be said that the wiki is not very well organized.

Since features, attributes and constraints are all handled by the compiler similarly to variables, their names cannot contain spaces for example. A naming scheme such as camel-cased names like in our model (figure 6) is needed. Shortening the names might be a good idea as well instead of just camel-casing full descriptions of them, because the latter is inconvenient to reference and makes the model look messy. The drawback of this is that the meaning of the features, attributes and constraints is hard to understand from looking at the model alone. There is probably need for some external descriptions of the clafers in many cases, and possibly even in our example context as well. Same variable names are allowed in different scopes, which seems to make sense for things such as 'maxSize' in our example, but there is a drawback to that approach, as discussed in the next section.

The instance generator that is directly accessible from the editor through a keyboard shortcut is useful for seeing whether the model you have made is indeed what you meant to do. This was the case with our model as well. Several times cycling between generated instances revealed that the structure or syntax of the model was wrong even though the model or syntax was not invalid. Even with a large amount of instances as with our model, cycling between them is surprisingly practical, as there is virtually no lag.

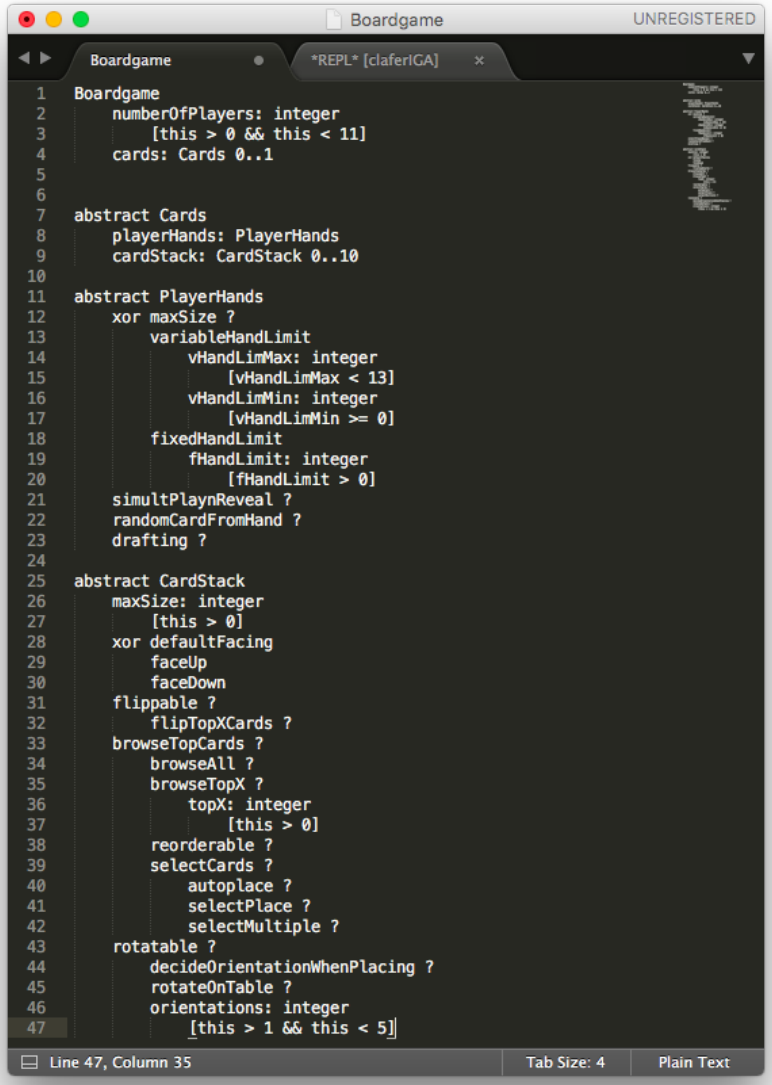

Figure 6: A generic model for a variety of board games

4.2.3 Product configuration

PV

Unfortunately opening the configuration perspective caused the whole variant management plugin to become completely unresponsive and consume all available processor power with PV version 4.0.9. No way to install an earlier version was found. Therefore, activities related to product configuration with became impossible to conduct with the tested setup.

CT

```
1  Poker: Boardgame
2      [minPlayers = 2]
3      [maxPlayers = 10]
4      [cards]
5      [playerHands.maxSize]
6      [fixedHandLimit]
7      [fHandLimit > 1 && fHandLimit < 6]
8      [simultPlaynReveal]
9      [!randomCardFromHand]
10     [!drafting]
11     [#cardStack = 1]
12     [cardStack.maxSize = 52]
13     [faceDown]
14     [!flippable]
15     [!browseTopCards]
16     [!rotatable]
17
```

Figure 7: A model of different poker games

```
Gameset1: Boardgame
    [maxPlayers = 4]
    [minPlayers = 3]
    [#cardStack = 1]
    [orientations = 2]
    [!playerHands.maxSize]
```

Figure 8: A fairly generic model of games with a card deck

With CT, there are two ways to do things related to product configuration. The first, which seems more prominent in the wiki and some other documentation, is putting the configuration specifications together with the model into the same file. This means defining concrete clafers that instantiate abstract ones, and removing some of their variability with constraints. The other way is using Clafer Configurator. Both the SublimeText integrated version and ClaferConfigurator were tested first

with a relatively complete (but simplified) configuration of poker games (figure 7), and then with a highly generic set of games (figure 8).

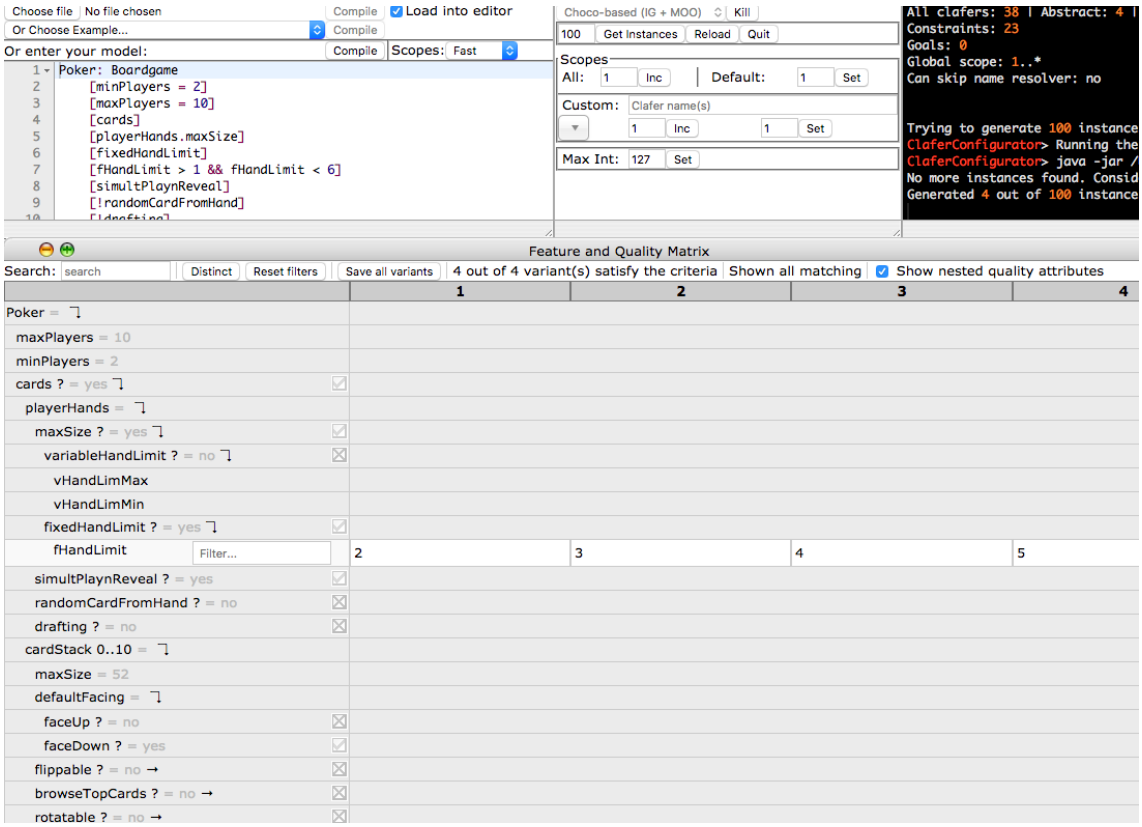

Figure 9: ClaferConfigurator

It was found out that starting from a completely unconfigured state, and that working towards a partial configuration is not practical with ClaferConfigurator due to a couple of reasons. First, it generates instances *depth first*, i.e. in such an order that similar features are as close as possible to each other. Secondly, the GUI for making configuration choices only filters the list of already generated instances and cannot affect the generation, as seen in figure 9. Thirdly, trying to generate any more than 1000 instances caused the web app to freeze indefinitely. These factors together mean that any model that could produce significantly more than 1000 product configurations cannot actually be configured through the GUI. If all generated instances have a certain selection, that selection will appear pre-made and unchangeable in the GUI. CC has an editor window, but no ability to save, and SublimeText is better suited for editing the textual model. However, once you have reached a stage of partial configuration, where it is possible to generate all the instances for it, CC becomes very useful, as it nicely visualizes the whole partial configuration at once,

and allows further configuration through an easy-to-use clickable interface.

Writing the configurations as constraints directly into the model is the only way to reach even partial configurations with CT. It works, but gets cumbersome and slow. When using non-unique variable names, such as maxSize in our case, they need to be referred to by prefixing the name with the scope: e.g. playerHands.maxSize.

The problem of workload resulting from editing the textual configurations can be eased to some extent by clever use of abstract clafers. Putting more generic configuration choices to abstract clafers with different levels of abstraction can make reuse of configuration definitions a bit cleaner and more convenient. But to make the abstract clafers actually useful for many products and sets of products would require a lot of up-front planning and thus that is really an option only with a very proactive approach to SPLE. In the example context, a more reactive approach is natural. Consequently, all the configuration choices are written inside a single abstract clafer corresponding a partial configuration.

The SPLE workflow is greatly hampered in CT by not being able to refer to clafers in other files. One has to either make multiple files and copy-paste the abstract clafers and some configuration choices between files or keep everything in same file, but comment out concrete clafers for other product sub-sets, or make every clafer that is not currently needed for instance generation abstract. The former is probably the only way to go when the scale of the SPLE project is as large as our full model (see drawn draft of the feature tree earlier), as otherwise the file would get inconveniently large. On the other hand, copy-pasting to different files has its own serious drawbacks as well.

Clafer can print instances into a text file with a format that could be read by some build tool. There are no such tools readily available, but it would not be too hard to make one for one's own project. The lists of instances can be saved both in CC and the SublimeText integration. In SublimeText, you can only save the instances generated so far (see figure 10), whereas in CC you can save just the instances that match the filtering choices done through the GUI, which is very useful. This is one of the reasons why it probably makes sense to use both.

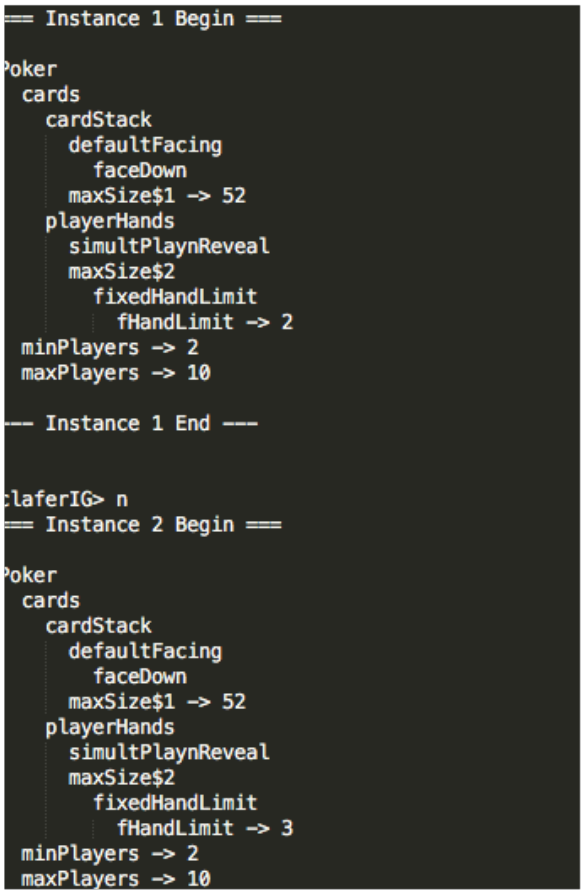

Figure 10: Poker game product instances

4.3 Summary of the results

Table 1 summarizes the support each tool provides for concepts and functions that were included in the evaluation based on the literature review. Three plus marks signifies good support for all important use cases and good usability. Two plus marks means that the concept or functionality is supported, but there are some usability problems or limitations with some specific important use cases. One plus mark (without *) means there is support, but it is very limited or there are major usability problems. One asterisk (*) means that the information is based on documentation due to differences between free and non-free versions. Two asterisks mean that support has been planned for a coming version.

Concept	PV	CT
Feature types		
Group cardinality	++	+++
Feature cardinality	-	++
Default features	+	-**
Constraints and attributes		
Simple constraints	+++	++
Complex constraints	++	++
Attributes	++	+++
Validation and error checking		
Validity checking	+	++
Detection of dead features	-	-
Error and warning messages	+	++
Inheritance and modularization		
References	+*	+
Inheritance	+*	++
Configuration		
Partial configurations	+++	+
Automatic generation	++	++
Visualization		
FD visualization	+++	-
Visualization of configurations	-	+

Table 1: Summary of each tools support for different concepts and functions

The information whether there is any support for an item is based solely on the first experiment, while the level of support is based on both, and is partly subjective.

In table 2 the subjective strengths and weaknesses of each tool are summarized based on the two experiments. A (+) is a strength, and (-) a weakness. When there is an (E) in the end of an item, it means that the claim or observation may not be generalizable, and is based on experiences with working on the example on a specific system, and assuming a specific context. Only the strengths and weaknesses that were observed in the two limited studies are listed. Things that were out of scope, and things that could not have been tested for various reasons are omitted.

Pure::Variants	**Clafer Tools**
Installation, integration and maintenance	
+ The software is quite mature and maintained relatively actively	+ Free open source software that can be tailored to one's own needs
+ Very easy to install and upgrade through Eclipse	+ Integrations for a fully textual tool should be easy to make
+ Existing integrations with many third party tools	
- Full version is not available for free	- Installation takes several steps and requires workarounds and manual tinkering on some systems (E)
- Some important functionality (modularization, inheritance, reuse) left out or unusable due to bugs in free version (E)	- ClaferConfigurator does not integrate well with the SublimeText plugin
- Dependency on a closed-source proprietary tool can be risky with very long term projects such as SPLs	- ClaferConfigurator's editor has no option to save model to a file
- Version 4.08 had a dependency on deprecated Eclipse version	- No notable integrations to other development and software project tools
- Version 4.09 made the tool unusable in practice on the tested system, and could not be rolled back (E)	

Adding and editing features and other basic modeling tasks

+ UI for basic feature addition is intuitive and easy to learn

+ Only necessary info shown to user

+ Human-readable names improve model comprehensibility (E)

+ Good support for CBFM style features

+ Helpful examples available online (E)

+ Efficient for advanced users

+ Restructuring the model is easy (E)

+ Printing product instances shows mistakes in models easily (E)

+ Low lag

- Lag in the UI together with the amount of pointing and clicking required reduce efficiency (E)

- Model can change unexpectedly without notice ('or' type features)

- No easy way to verify sanity of the model (E)

- Steep learning curve, especially without development background

- In documentation and examples, different syntax is used for same or similar things, reducing their usefulness for learning

- Everything is always a variable, and a tradeoff between low comprehensibility and high verbosity of names has to be made

- Requires learning a new language

Constraints, attributes and advanced modeling techniques

+ Easy and convenient support for default features	+ Syntax for constraints and attributes is concise and simple, but still powerful
+ Relations easy to do via the GUI and can cover many needs for constraints	+ It was easy to manage the many attributes in the example model (E)
+ Helpful info about constraints and attributes easy to find from manual	
+ Softer constraints that only cause warnings instead of errors or forced selection can be done with restrictions	
- Non-relation constraints are cumbersome and difficult to add and manage	- No default features
- Attributes handled in an unnecessarily complex way for the example (E)	

Scalability and comprehensibility for different stakeholders

+ Intuitive and comprehensible visual syntax	+ Abstract clafers can be a good tool for modularization and enabling reuse of model code
+ Graph view of the model seems useful for information transfer (E)	

- Modularization and inheritance techniques did not work at all with the tested PV versions (E)	- Everything goes into a single file, reducing usefulness of abstract clafers - References are hard to use or even understand in practice and documentation was not very helpful - Abstract clafers take practice to utilize effectively
Product configuration and error checking	
+ Customizable automatic error checking, can be switched to manual + Easy to do configuration choices by clicking checkboxes and by clicking and typing into text fields + Changes to FM dynamically reflected to configuration view + Easy to switch back and forth between FM and configuration view by switching editor tabs	+ Easy printing of product instances helps with a light-weight approach to SPLE (E) + ClaferConfigurator is good for visualizing partial configurations + Partial configurations can be made more complete by clicking checkboxes in CC's GUI + Compiler errors are easy to understand and locate (E)

- Automatic model checking and validation can make the UI unresponsive (E)	- Order of instances getting generated cannot be customized
- Checking sanity of the FM requires clicking through configuration choices (E)	- Exclusion of instances from being generated has to be done through constraints in textual model, not CC's GUI
- No detection of dead features	- CC freezes without recovery on tested system when generating over 1000 instances (E)
- The configuration generation process can be too heavy-weight when code is not generated (E)	- Reaching a partial configuration specific enough for CC in practice takes lots of typing and copy-pasting
	- Working on configurations in the same file that has all the abstract clafers is error-prone and leads to poor user experience (E)

Table 2: Subjective strengths and weaknesses of each tool per usage scenario

5 Discussion

Many of the strengths and weaknesses of each tool seem to be related to their architectural design choices related to the decision to be either graphical or textual. For example, the user experience for everything requiring textual definitions is fairly poor in PV, whereas having to put all information related to both the model and configurations into a text-file severely hurts CT's usability. Additionally the GUI elements provided by Eclipse may not always be very well suited for SPLE, and the lag in the GUI was often a problem in the tested setup. Sometimes working on text would be faster. Thus it seems like an ideal tool might be text-based with a language such as Clafer, but with a complete GUI that can manipulate the underlying textual models.

5.1 Answer to RQ1

RQ1: What are the most important requirements for SPLE tools for a small-to-medium sized organisation aiming to experiment with SPLE? The evaluated functionalities and properties and the evaluation criteria used (table 1) were picked from literature with this context in mind. The experiments did not prove their importance. On the other hand, not many candidates for other requirements came to mind while performing the studies, nor did any of the assessed characteristics seem useless. Detection of dead features was not found important however, and it could be dropped from the requirements.

5.2 Answer to RQ2

RQ2: How well those requirements are met in two specific SPLE tools, Pure::Variants and Clafer tools? The list of requirements that was picked from literature for this study was supported by both tools quite well. There was one piece of functionality that was well supported on PV, but not at all on CT: FD visualization. Although it may be harder to do for a purely text-based tool, it could be argued that it is especially needed there due to a textual representation being generally harder to decipher for non-coders than a graphical one. On the other hand, PV had no visualization of configurations, whereas CT could be considered to have some support for it via Clafer Configurator.

When it comes to partial configurations, Clafer Configurator could be a powerful tool

for this, but its inability to rule out or force configuration choices graphically instead of just filtering already generated configurations make it impossible to do partial configurations with it, except for models that generate relatively tiny amounts of valid product instances. On the other hand, PV's configuration view is naturally suited for partial configurations.

The support for modularization and inheritance techniques was found either lacking in usability or reliability in both tools, although this is likely to be fixed for PV's commercial version. Therefore increasing scale and complexity of models can quickly become a problem with both CT and PV (community version). Especially having to put everything, including partial configurations and abstract clafers (similar to classes that are instantiated), into a single file makes coping with increasing complexity and scale of the model unnecessarily hard.

5.3 Answer to RQ3

RQ3: How do the studied tools compare against each other when it comes to their suitability for the chosen context (a digital board game platform)? The context-specific study was left very limited in scope due to not being able to develop the model for the board game SPLE any further with PV. Additionally many tool functionalities could not be tested with PV community edition. For these reasons only very basic features of the tools could be properly compared in terms of actual usage.

Both tools were found to have their own strengths and weaknesses. CT works well with models that generate small enough numbers of configurations for Clafer Configurator to be usable without very extensive textual configuration before loading the model in CC. Conversely, PV may be a better choice with models that generate vast amount of possible configurations due to numerical attributes, like in our example, or the sheer size of the model. On the other hand, PV seems best suited to projects where there is heavy commitment to SPLE and up-front planning. CT being free software and its ability to quickly and conveniently show individual product configurations is useful when working on the model iteratively or when simply trying out SPLE. One more factor to consider are the skill-sets of people working on the project. Both tools might be hard to use without a technical background, but CT is especially difficult to get started with without a coding background, and it doesn't really have functionality to effectively communicate to non-technical stakeholders either.

One more thing to consider is the operating system and other issues related to installation of the tools, as one version (the latest version as of the time of testing but not as of writing this) of PV was practically unusable on two different versions of OSX.

5.4 Answer to RQ4

RQ4: How common requirements for SPL tools can be generalized to be applicable for both graphical and text-based tools? Based on the experiments, the same requirements (see answer to RQ1 above) can be reasonably used for both graphical and textual tools. As mentioned in the section for RQ2 above, the levels of support for each functionality or other characteristic was very similar with both tools. From this it would seem like the chosen requirements are not particularly biased towards either GUI-based or text-based tools. There was a big difference in regards to support for FD visualization, but text-based tools would benefit greatly from visualization as well. On the other hand, graphical tools require the user to do text editing in some places. Therefore both purely textual or purely graphical tools would be inherently limited.

6 Conclusions

Pure::Variants and Clafer tools are both rather complete tools with comparable support for various aspects of SPLE commonly talked about in literature. For the context of a small or medium sized organization getting started with or trying out SPLE, both tools have their own strengths and weaknesses, and the best choice for a tool probably depends on a number of factors. Some such factors could be size of the model and approximate number of valid configurations. CT has poor support for increasing scale and complexity. The same can be said about partial configuration, which would especially be needed with larger and more complex models. PV on the other hand had some issues with reliability and stability on the tested setup (OS X). Additionally the commercial version seems to be needed to gain access to any modularization and inheritance techniques.

For the example domain chosen for this thesis, board game implementations, not many specific issues were found or observations made due to running into technical problems and time limitations. One notable thing found was that there seem to be many places were numerical attributes can be used. If they are used a lot, at least Clafer tools becomes tedious to use due to exponentially increasing numbers of potential configurations. With PV it was not possible to test configuration at all due to what is probably a bug. For modeling tasks, both tools performed fairly well and there was no big difference.

Based on looking into the two tools in this study, both the text-based and GUI-based approaches have their own weaknesses. How to combine the best characteristics of both would be an interesting area of future research, and something to consider for tool developers as well.

The comparison in this work was rather limited and highly subjective, and more quantitative and scientifically sound comparisons of these two tools remain an interesting, yet not much explored subject for future research. The same can be said about board games as an application domain for SPLE.

References

ABM+13 Antkiewicz, M., Bąk, K., Murashkin, A., Olaechea, R., Liang, J. H. J. and Czarnecki, K., Clafer tools for product line engineering. *Proceedings*

of the 17th International Software Product Line Conference co-located workshops. ACM, 2013, pages 130–135.

ABMG12 Asadi, M., Bagheri, E., Mohabbati, B. and Gašević, D., Requirements engineering in feature oriented software product lines: an initial analytical study. *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM, 2012, pages 36–44.

ANAV10 Alves, V., Niu, N., Alves, C. and Valença, G., Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, 52,8(2010), pages 806–820.

ASG+14 Asadi, M., Soltani, S., Gasevic, D., Hatala, M. and Bagheri, E., Toward automated feature model configuration with optimizing non-functional requirements. *Information and Software Technology*, 56,9(2014), pages 1144–1165.

BDA+14 Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K. and Wąsowski, A., Clafer: unifying class and feature modeling. *Software & Systems Modeling*, pages 1–35.

BE13a Bagheri, E. and Ensan, F., Light-weight software product lines for small and medium-sized enterprises (smes). *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13. IBM Corp., 2013, pages 311–324.

BE13b Bagheri, E. and Ensan, F., Light-weight software product lines for small and medium-sized enterprises (smes). *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2013, pages 311–324.

Ber12 Berger, T., Variability modeling in the wild. *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM, 2012, pages 233–241.

BNR+14 Berger, T., Nair, D., Rublack, R., Atlee, J. M., Czarnecki, K. and Wąsowski, A., Three cases of feature-based variability modeling in industry. *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pages 302–319.

BRN+13 Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K. and Wąsowski, A., A survey of variability modeling in industrial practice. *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2013, page 7.

BSL+13 Berger, T., She, S., Lotufo, R., Wasowski, A. and Czarnecki, K., A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39,12(2013), pages 1611–1640.

CB11 Chen, L. and Babar, M. A., A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53,4(2011), pages 344–362.

CGR+12 Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K. and Wąsowski, A., Cool features and tough decisions: a comparison of variability modeling approaches. *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*. ACM, 2012, pages 173–182.

CHE04 Czarnecki, K., Helsen, S. and Eisenecker, U., Staged configuration using feature models. *International Conference on Software Product Lines*. Springer, 2004, pages 266–283.

CL16 Chimalakonda, S. and Lee, D. H., On the evolution of software and systems product line standards. *ACM SIGSOFT Software Engineering Notes*, 41,3(2016), pages 27–30.

CPP+16 Constantino, K., Pereira, J. A., Padilha, J., Vasconcelos, P. and Figueiredo, E., An empirical study of two software product line tools. Federal University of Minas Gerais, Brazil, 2016.

CSD07 Capilla, R., Sánchez, A. and Dueñas, J. C., An analysis of variability modeling and management tools for product line development. *Software and Service Variability Management Workshop-Concepts, Models, and Tools*, 2007, pages 32–47.

DFE14 Derakhshanmanesh, M., Fox, J. and Ebert, J., Requirements-driven incremental adoption of variability management techniques and tools: an industrial experience report. *Requirements Engineering*, 19,4(2014), pages 333–354.

DSF07 Djebbi, O., Salinesi, C. and Fanmuy, G., Industry survey of product lines management tools: Requirements, qualities and open issues. *15th IEEE International Requirements Engineering Conference (RE 2007)*. IEEE, 2007, pages 301–306.

DSNO+14 Da Silva, I. F., Neto, P. A. d. M. S., OĹeary, P., De Almeida, E. S. and de Lemos Meira, S. R., Software product line scoping and requirements engineering in a small and medium-sized enterprise: An industrial case study. *Journal of Systems and Software*, 88, pages 189–206.

EBB05 Eriksson, M., Börstler, J. and Borg, K., The pluss approach–domain modeling with features, use cases and use case realizations. *International Conference on Software Product Lines*. Springer, 2005, pages 33–44.

EDDT11 El Dammagh, M. and De Troyer, O., Feature modeling tools: evaluation and lessons learned. *International Conference on Conceptual Modeling*. Springer, 2011, pages 120–129.

EKS13 Eichelberger, H., Kröher, C. and Schmid, K., An analysis of variability modeling concepts: Expressiveness vs. analyzability. *International Conference on Software Reuse*. Springer, 2013, pages 32–48.

ES13 Eichelberger, H. and Schmid, K., A systematic analysis of textual variability modeling languages. *Proceedings of the 17th International Software Product Line Conference*. ACM, 2013, pages 12–21.

FLD+15 Fang, M., Leyh, G., Doerr, J., Elsner, C. and Zhao, J., Towards model-based derivation of systems in the industrial automation domain. *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pages 283–292.

HGR12 Holl, G., Grünbacher, P. and Rabiser, R., A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology*, 54,8(2012), pages 828–852.

IEC IEC, I., 25010 (2011). systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. *International Organization for Standardization, Geneva, Switzerland*.

KAT16 Kowal, M., Ananieva, S. and Thüm, T., Explaining anomalies in feature models. *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, 2016, pages 132–143.

LVV04 Lago, P. and Van Vliet, H., Observations from the recovery of a software product family. *International Conference on Software Product Lines*. Springer, 2004, pages 214–227.

NdCMM+11 Neto, P. A. d. M. S., do Carmo Machado, I., McGregor, J. D., De Almeida, E. S. and de Lemos Meira, S. R., A systematic mapping study of software product lines testing. *Information and Software Technology*, 53,5(2011), pages 407–423.

OMTR12 OĹeary, P., McCaffery, F., Thiel, S. and Richardson, I., An agile process model for product derivation in software product line engineering. *Journal of Software: Evolution and Process*, 24,5(2012), pages 561–571.

PB12 Pleuss, A. and Botterweck, G., Visualization of variability and configuration options. *International Journal on Software Tools for Technology Transfer*, 14,5(2012), pages 497–510.

PBvDL05 Pohl, K., Böckle, G. and van Der Linden, F. J., *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

PCF15 Pereira, J. A., Constantino, K. and Figueiredo, E., A systematic literature review of software product line management tools. *International Conference on Software Reuse*. Springer, 2015, pages 73–89.

PRB11 Pleuss, A., Rabiser, R. and Botterweck, G., Visualization techniques for application in interactive product configuration. *Proceedings of the 15th International Software Product Line Conference, Volume 2*. ACM, 2011, page 22.

PSF+13 Pereira, J. A., Souza, C., Figueiredo, E., Abilio, R., Vale, G. and Costa, H. A. X., Software variability management: An exploratory study with two feature modeling tools. *Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on*. IEEE, 2013, pages 20–29.

PV04 pure-systems GmbH, Variant management with pure::variants. Technical white paper, pure-systems GmbH, Magdeburg, Germany, 2004.

PVMAN pure-systems GmbH, *pure::variants User's Guide*, 2016.

RFBCRC09 Roos Frantz, F., Benavides Cuevas, D. F. and Ruiz Cortés, A., Feature model to orthogonal variability model transformation towards interoperability between tools. *Knowledge Industry Survival Strategy Initiative, Kiss Workshop ASE2009, Auckland, New Zealand*, 2009.

RGD10 Rabiser, R., Grünbacher, P. and Dhungana, D., Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52,3(2010), pages 324–346.

SBKM09 Savolainen, J., Bosch, J., Kuusela, J. and Männistö, T., Default values for improved product line management. *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 2009, pages 51–60.

SBSQ11 Simmonds, J., Bastarrica, M., Silvestre, L. and Quispe, A., Analyzing methodologies and tools for specifying variability in software processes. *Universidad de Chile, Santiago, Chile*.

SD07 Sinnema, M. and Deelstra, S., Classifying variability modeling techniques. *Information and Software Technology*, 49,7(2007), pages 717–739.

SRM11 Savolainen, J., Raatikainen, M. and Männistö, T., Eight practical considerations in applying feature modeling for product lines. *International Conference on Software Reuse*. Springer, 2011, pages 192–206.

SSAIEA16 Saeed, M., Saleh, F., Al-Insaif, S. and El-Attar, M., Empirical validating the cognitive effectiveness of a new feature diagrams visual syntax. *Information and Software Technology*, 71, pages 1–26.

STB+04 Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W. and Ferber, S., Introducing pla at bosch gasoline systems: Experiences and practices. *International Conference on Software Product Lines*. Springer, 2004, pages 34–50.

VdLSR07 Van der Linden, F. J., Schmid, K. and Rommes, E., *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.